PG12 乱数によるシミュレーション

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

numpy には特定の分布に従った乱数を生成するコマンドがさまざま用意されているが、 基本は区間[0,1]上の一様乱数である。 ここでは、この一様乱数を使ってコイン投げのシミュレーションをするとともに、簡単なプログラミングを学ぶ。

1. 一様乱数の生成

In [2]:

```
# 0以上1未満の乱数を1個生成する
np.random.rand()
```

Out[2]:

0.058941764234077265

In [3]:

```
for _ in range(5):
    print(np.random.rand())
```

0.24776695575909446
0.8972062120995984
0.5196273939870397
0.19973135774770034
0.09295266415393

実行するたびに異なる値が出力される. 計算機が作り出す乱数は, 見かけ上ランダムに数が並ぶような上手なアルゴリズムで 生成されるので, そのシードと呼ばれる初期設定が同じであれば当然同じ結果になる. ふつう乱数を使う場面では, いつも異なるばらばらの数列が欲しいので, システム時刻などを利用して、実行のたびに異なるシードが自動的に設定される. 一方で, プログラムの動作を確認したいときなどは, 同じ乱数を繰り返し使いたいこともある. そのときは, 乱数のシードを指定することで, 発生する乱数列を固定することができる.

In [4]:

```
np.random.seed(123)
for _ in range(5):
    print(np.random.rand())
```

0.6964691855978616
0.28613933495037946
0.2268514535642031
0.5513147690828912
0.7194689697855631

In [5]:

```
np.random.rand(5)
```

Out[5]:

```
array([0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752])
```

このように乱数のシードを固定していると、いつも同じ乱数列が発生する。

np.random.seed()

によって、乱数のシードの指定が解除される。

In [6]:

```
np.random.seed()
for _ in range(5):
    print(np.random.rand())
```

```
0.9268460430518108
0.40120094377836046
0.9412972356167275
0.37954363661410484
0.8171440320802191
```

2. 乱数列を指定したサイズの NumPy アレイで返す

In [7]:

```
# 1次元配列
np.random.rand(4)
```

Out[7]:

```
array([0.1369794 , 0.76697019, 0.12400219, 0.29873035])
```

In [8]:

```
# 4変量データとして、3個生成する。（3×4　行列）
np.random.rand(3,4)
```

Out[8]:

```
array([[0.2340403 , 0.13282837, 0.66788872, 0.51177461],
       [0.78128542, 0.96869582, 0.25170972, 0.25443081],
       [0.52833437, 0.20615633, 0.81456406, 0.35682471]])
```

In [9]:

```
# 4変量データを3個生成して、それを1組として2組を生成（3×4行列を2個生成）
np.random.rand(2,3,4)
```

Out[9]:

```
array([[[0.06782021, 0.78306633, 0.69402898, 0.93428831],
        [0.01099127, 0.939181  , 0.87621056, 0.0164395 ],
        [0.89532934, 0.93137979, 0.10601757, 0.85471411]],

       [[0.43399125, 0.29936064, 0.88738484, 0.84691126],
        [0.84891407, 0.59028668, 0.37055309, 0.01215443],
        [0.63060145, 0.39851151, 0.66925946, 0.08218952]]])
```

3. 乱数の分布の確認

In [10]:

```
trial = 10000                  # 発生させる乱数の個数
rn = np.random.rand(trial)     # trial 個の乱数を発生させてアレイにする
rn
```

Out[10]:

```
array([0.55509545, 0.27204022, 0.25430345, ..., 0.65029503, 0.59237051,
       0.77145597])
```

NumPy アレイを DataFrame の形式にしたければ、

rndf = pd.DataFrame(rn)

とすれば、rndf と言う名前のDataFrameとして扱える。

In [11]:

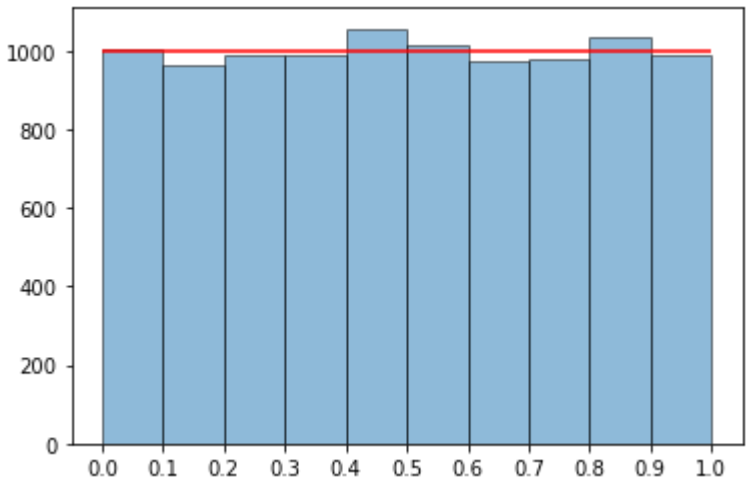
```
# rn をもとにヒストグラムを作る
plt.hist(rn, range=(0, 1), bins=10, alpha=0.5, ec='k')

plt.hlines(trial/10, 0, 1, color='red')    # y=trial/10 の水平線
plt.xticks(np.arange(0, 1.1, 0.1))         # x軸の目盛

#plt.savefig('D:¥¥2022Stat_Python¥¥Data¥¥rn_hist.eps')    # 図の保存
```

Out[11]:

```
([<matplotlib.axis.XTick at 0x23c8e749df0>,
  <matplotlib.axis.XTick at 0x23c8e749dc0>,
  <matplotlib.axis.XTick at 0x23c8e73c910>,
  <matplotlib.axis.XTick at 0x23c8ea06550>,
  <matplotlib.axis.XTick at 0x23c8ea06a60>,
  <matplotlib.axis.XTick at 0x23c8ea065e0>,
  <matplotlib.axis.XTick at 0x23c8ea0f160>,
  <matplotlib.axis.XTick at 0x23c8ea0f670>,
  <matplotlib.axis.XTick at 0x23c8ea0fb80>,
  <matplotlib.axis.XTick at 0x23c8ea140d0>,
  <matplotlib.axis.XTick at 0x23c8ea145e0>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



基本統計量を計算して理論値と比較しておこう。

理論値：平均値=0.5, 分散=1/12=0.0833, 標準偏差=√3/6=0.2886, 最小値=0, 最大値=1

In [12]:

```
# 基本統計量
rn.mean(), rn.var(), rn.std(),  rn.min(), rn.max()
```

Out[12]:

```
(0.5009762803307658,
 0.08269847484293683,
 0.28757342513336803,
 4.4085668084337115e-05,
 0.9998805736216176)
```

上のコードは「メソッド」と呼ばれる形式である。関数の形式も使える。

In [13]:

```
np.mean(rn), np.var(rn), np.std(rn), min(rn), max(rn)
```

Out[13]:

```
(0.5009762803307658,
 0.08269847484293683,
 0.28757342513336803,
 4.4085668084337115e-05,
 0.9998805736216176)
```

4. 正規乱数

In [14]:

```
trial = 10000
mean = 0
std = 2
data = np.random.normal(mean, std, size=trial)
data
```

Out[14]:

```
array([-0.197975  ,  2.15687496, -0.57238446, ..., -0.31377178,
        2.16538055, -2.82044104])
```

In [15]:

```
min(data), max(data)
```

Out[15]:

```
(-9.065867152901397, 7.394550605264897)
```

In [16]:

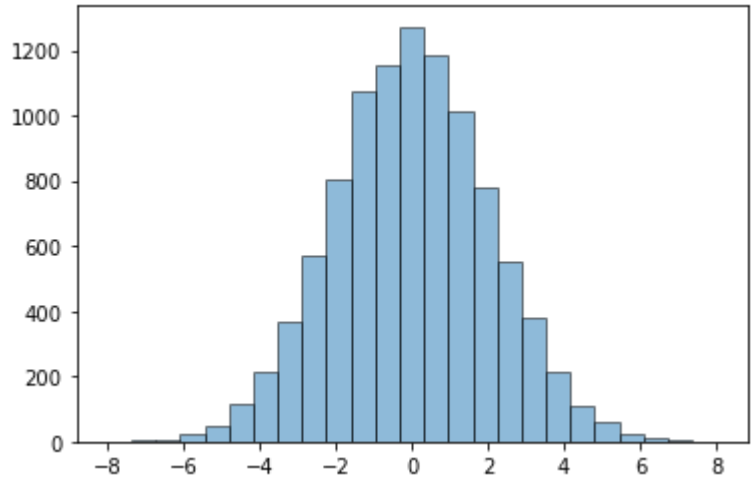
```
np.mean(data), np.var(data), np.std(data)
```

Out[16]:

```
(-0.006431957864918466, 4.140705637974669, 2.034872388621623)
```

In [17]:

```
plt.hist(data, range=(-8, 8), bins=25, alpha=0.5, ec='k')
plt.show()
```



正規分布の密度関数と比較しよう。 密度関数の定義式を使って ndf(x) という名前で導入する。

In [18]:

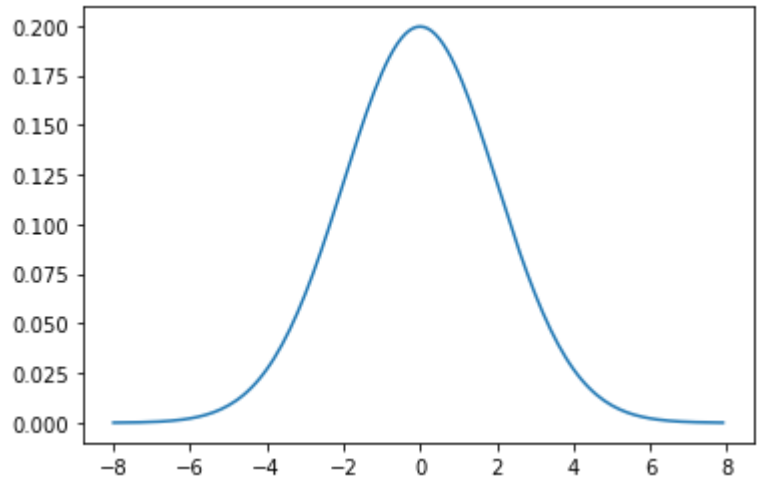
```
def ndf(x):
    return 1/np.sqrt(2*np.pi*std**2)*np.exp(-(x-mean)**2/(2*std**2))
```

In [19]:

```
x_range=np.arange(-8, 8, 0.1)
plt.plot(x_range, ndf(x_range))
```

Out[19]:

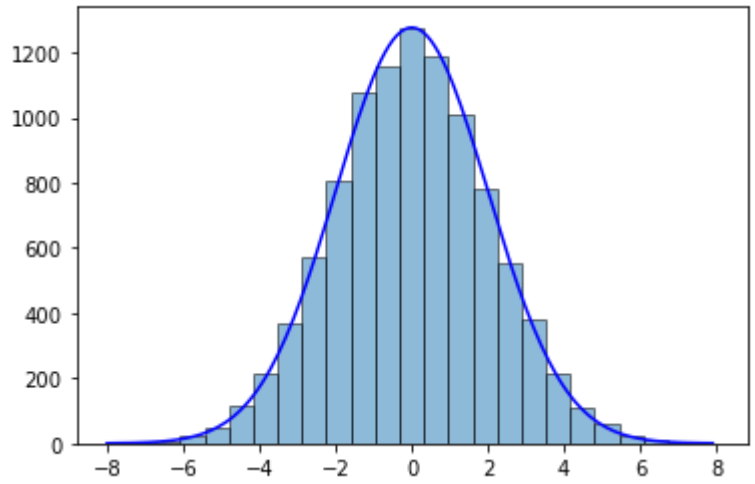
```
[<matplotlib.lines.Line2D at 0x23c8eb82b80>]
```



ヒストグラムと密度関数のグラフを重ねて比較しよう。 ヒストグラムでは x 軸の範囲 $-8 \leq x \leq 8$ を25等分したので、ヒストグラムの1個の長方形の横の長さは 16/25 縦の長さは 1 である。全度数は trial 個なので、ヒストグラムの作る面積は 16/25× trial となる。 一方、密度関数のグラフの作る面積は 1 であるから、密度関数の値を16/25× trial 倍して、初めて比較可能になる。

In [20]:

```
plt.hist(data, range=(-8, 8), bins=25, alpha=0.5, ec='k')
x_range = np.arange(-8, 8, 0.1)
plt.plot(x_range, 16/25 * trial * ndf(x_range), color='blue')
plt.show()
```



In [ ]:

```

```